Delft University of Technology

Bio-Inspired Intelligence and Learning for Aerospace Applications
AE4350

Learning to Slither:

Applying Deep Q-Learning to the Snake Game

Lars van Pelt 5629632

August 16, 2024

Summary

This report investigates the application of Deep Q-Learning, a powerful combination of Q-learning and deep neural networks, to the classic Snake game. The study focuses on the implementation of a Deep Q-Network and subsequently evaluates the agent's performances and sensitivity under various parameter settings. Relevant varied parameters are the epsilon decay rate, discount factor, learning rate, hidden layer size, and batch size. The findings of this study highlight the importance of tuning these parameters such that a balance between agent exploration and exploitation is found, leading to the best overall agent performance within this study's constraints.

The agent performance analyses show that epsilon decay rate, discount factor, and learning rate each affect the agent behaviour and learning stability. For enhancing performance, it was found that the epsilon decay rate and discount factor should be tweaked to 0.96 and 0.85, respectively. Contrarily, the initial learning rate of 0.001 is found to yield acceptable results. Additionally, the neural network architecture was found to affect the agent's ability to learn complex patterns as well, though increasing the number of neurons beyond 128 neurons offered diminishing returns and so is found to suffice. Variations in batch size revealed a trade-off between learning speed and stability, with generally larger batches leading to more reliable performance.

As such, the report underlines the critical role of parameter tuning in reinforcement learning applications and illustrates the potential of Deep Q-Learning in dynamic, game-based environments such as the Snake game. Despite the insightful results, the study faced limitations due to hardware constraints and time constraints, which restricted the sessions to 2000 episodes. This limitation likely prevented the agent from fully converging to optimal strategies, and so it is suggested that future research with extended training sessions could further enhance performance.

Contents

List of Acronyms

AI	Artificial Intelligence
DQN	Deep Q-Network
MSE	Mean Squared Error
ML	Machine Learning
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
TD	Temporal Difference

Nomenclature

α	Learning rate
ϵ	Exploration probability
γ	Discount factor
π	Action selection policy
$\sigma(x)$	Sigmoidal activation function
θ_i	Neural network weights at iteration i
θ_i^-	Target neural network weights at iteration i
A_t	Agent action at timestep t
$L_i(\theta_i)$	Loss function
$Q(S_t, A_t)$	Q-value corresponding to state-action pair
R_{t+1}	Reward signal
S_t	Environment state at timestep t
S_{t+1}	Environment state at next timestep t+1
$V(S_t)$	Value function at current state
$V(S_{t+1})$	Value function at next state

1 Introduction

In contemporary industries, Artificial Intelligence (AI) has managed to establish itself firmly. Its capabilities now presented to the public cause the industry to pursuit AI to master even more complex tasks, which has led to significant advancements in Machine Learning (ML). Various techniques contributing to this have emerged, among which Deep Q-Learning has proven to be a powerful method for training AI agents in dynamic decision-making.

The Deep Q-learning method is a form of Reinforcement Learning (RL) which utilises a combination of the Q-learning method with deep neural networks. Q-learning is used for teaching the agent to learn policies the way humans do; by receiving rewards when doing things right and receiving corrections when making mistakes. On the other hand, deep neural networks are primarily used for handling complex spaces and recognising patterns in them. Leveraging the combination of both methods allows for teaching AI agents decision-making policies through independent training, reducing the need for additional human input in the form of feature engineering. The potential the application of this method offers has been demonstrated to a variety of Atari 2600 games as well as the Chinese strategy board game "Go", marking it an interesting point for AI research. [1][2]

This report explores the application of Deep Q-Learning Method to the classic Snake video game. Snake is a genre of video games which was developed in the early days of video gaming on electronic devices. It offers the player the challenge of growing the snake by eating bits of food without colliding into either walls or the snake's body. While this does seem like a simple task for an experienced human player, it may not be as simple for an AI agent that has no prior knowledge. In order to be successful, the agent needs to learn from scratch how to conduct path finding, avoid obstacles, and plan its next moves ahead of its current state. These characteristics therefore make the snake game an interesting framework to which Deep Q-Learning can be applied, such that broader understanding of the method may be obtained. As such, this report starts with a description of the applied methods in Chapter 2. Following this, the results of training the agent using this method are presented in Chapter 3. A discussion on both is provided in Chapter 4 and, lastly, the conclusions of this report are presented in Chapter 5. All code relevant to this study can be retrieved at: Link to GitHub Repository.

2 Methodology

2.1 Reinforcement Learning

The section of ML which entails agent's decision-making logic in an environment such that it generates cumulative rewards is called Reinforcement Learning (RL). Essentially, this form of ML considers teaching an agent to learn good behaviour in a way similar to humans do. That is, through exploring an environment based on trial-and-error principle, in which good decisions translate to positive rewards and errors to negative rewards. The way the agent interacts with its environment can be framed by the Markov decision process, a cornerstone in RL and visualised in Figure 1. [3][4]

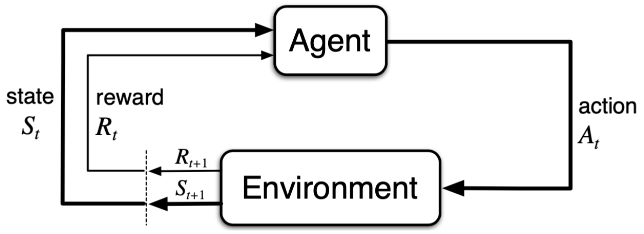


Figure 1: Markov decision process of agent-environment interaction

In this process, the interaction is discretised by timesteps, t = 0,1,2,3... Each timestep, the agent is presented with a state of the environment (S_t) it is in. Based on this state, the agent is to decide on an action (A_t), which affects the environment that is transitioned to the next timestep (S_{t+1}) and yields a numerical, pre-defined reward (R_{t+1}). The reward signal serves as direct feedback for the agent, telling it what is good and what not immediately. Moreover, the way the agent selects which action to take depends on which policy is applied and so there are various strategies available. This, together with the reward signal, a value function telling the agent what is desirable in the long-term, and optionally a model of the environment, form the main elements of RL. [4][5]

2.2 Temporal difference updating

One of the novel ideas in RL is Temporal Difference (TD) updating. The main principle of TD updating relies on considering differences between intermediate estimations of value functions, unlike Monte-Carlo methods that rely on one-time, full episode updates. In other words, the learning process is based on trying to match the current prediction of current input closely to that of the next prediction at the next step. Mathematically, this principle can be denoted by Equation 1. [6]

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{1}$$

Here, the estimated value of the value function at a certain state, $V(S_t)$, is dependent on the estimated increment to the value function. This estimated increment is computed based on a number of factors, being the current value of the value function, the expected reward and value of the value function at the next timestep, R_{t+1} and $V(S_{t+1})$, respectively, as well as the learning rate α and discount factor γ. Here, the learning rate and discount factor are hyper-parameters that may be adjusted to either influence how much new information influences each value function update or to influence the importance of future rewards, respectively. By means of this principle, TD therefore learns to update estimates based on other estimates, a method called Bootstrapping. [4][6]

2.3 Q-Learning

The Q-learning method is a revolutionary RL method developed in 1989, in which the agent learns decision-making policies in Markovian domains in a model-free matter. That is to say that the agent learns the environment's dynamics solely from trial-and-error, utilising TD updating. Contrary to TD updating, the interaction of the agent with the environment is denoted by state-action pairs, rather than states only. Here, an agent's action A_t at a certain state S_t corresponds to a certain "Quality" Q, which in the Q-learning algorithm serves as the value function $Q(S_t, A_t)$. Now, the Q-learning algorithm itself is introduced by Equation 2. [4]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2}$$

This algorithm works iteratively based on the visualisation of Figure 2, which is mostly initialised with Q-values

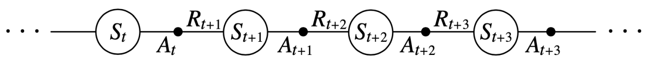


Figure 2: General updating course of Q-Learning episodes

set to zero. For this, the algorithm must assume all possible state-action pairs and so a Q-table of rows equal to number of states and columns equal to number of actions per state, filled with zeros, is to be set up. Utilising this table, the algorithm updates it repeatedly as the agent initially explores the environment and observes the corresponding rewards. By considering maximum Q-values, i.e., maximum quality, and by iteratively updating the Q-table accordingly, the algorithm is able to converge to the optimal policy (Figure 3) for decision-making, satisfying the Bellman equation (Equation 3). [4][7]

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t + \gamma \max_a Q(s_{t+1}, a_{t+1})|S_t = s, A_t = a] \tag{3}$$

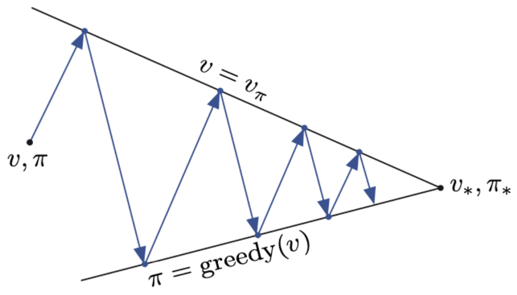


Figure 3: Convergence process Q-learning

The rate and optimality of this convergence is, however, largely dependent on the policy π the agent uses to base its selection of actions on. A critical condition for convergence is that all state-action pairs must be visited multiple times. This implies that a good balance must be found for letting the agent explore (trying new actions) versus letting it exploit (choosing actions with largest rewards); a known dilemma in Q-Learning. The most common strategy for managing this is the ϵ-greedy policy, where the agent explores with decaying probability ϵ. When a pre-defined threshold value for ϵ is reached, the agent starts to exploit instead. Managing ϵ and its decay rate properly therefore contribute to proper convergence of the algorithm. [4][7]

2.4 Deep Q-Learning

Deep Q-Learning is an extension of the classic Q-Learning method that leverages the application of it to a neural network; a Deep Q-Network (DQN). Classic Q-learning is known to struggle with significantly large state-spaces due to the dimensionality properties of its Q-table. By using a neural network with weighted links, one relies on non-linear function approximation instead, meaning that through continuous representation the tabular state-space issues are mitigated. A general form of a neural network is a feed-forward network as shown in Figure 4. [4]

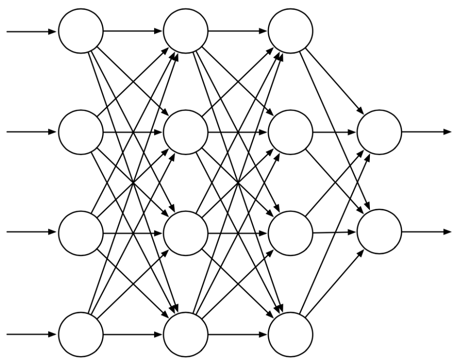


Figure 4: Generic feed-forward neural network

In Figure 4, the layers are defined from left to right as: input layer, 2 "hidden" layers, and output layer. Here, the input layer takes the state of the environment, whereas the output layer generates Q-values relating to each possible action in that state. Additionally, the middle layers are called hidden layers as they neither serve as input layer nor output layer. The hidden layers in a feed-forward network are to be considered its workhorses, as they enable the network to learn complex patterns in the data. In fact, a network with a single hidden layer containing a sigmoidal activation function can approximate any continuous, non-linear function on a compact region of the network's input space arbitrarily well. Evidently, the activation functions (represented by circles in Figure 4) are paramount for "universal" function approximation with neural networks. One of the most common activation functions used is the Rectified Linear Unit (ReLU) function, presented in Equation 4. [8][9]

$$\sigma(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Since the design of hidden units is a complex and extensive field of research, ReLU functions serve as a default choice for activation function. Due to their similarity to linear units, ReLU functions are easy to implement and reduce the impact on computational load. They furthermore introduce sparsity by keeping negative inputs zero, boosting convergence of the algorithm. While not a universal solution, ReLU functions have shown in practice to deliver sufficient model accuracy and thus great overall performance. [1][10]

The function approximation capabilities can be utilised to approximate the Q-value function and so the neural network parameter θ_i, representing the network weights at iteration i, is added. Hence, the Q-value function becomes $Q(S_t, A_t, \theta_t)$. The target Q-values are determined through Equation 3 and are stored in a separate target network $Q(S_t, A_t; \theta_i^-)$. For learning policies, theDQN objective becomes minimising the differences between predicted Q-values and target Q-values. Furthermore, during training, the agent's experiences are stored as (S_t, A_t, R_t, S_{t+1}) in a dataset called the experience replay buffer. This is then used for learning purposes, as batches with pre-defined batch size are drawn uniformly, to which then Q-learning updates are applied using the loss function of Equation 5. By ultimately minimising the loss function, and so, the Mean Squared Error (MSE), convergence of the model is attained.

$$L_i(\theta_i) = \mathbb{E}[(R_t + \gamma \max Q(S_{t+1}, A_{t+1}; \theta_i^-) - Q(S_t, A_t; \theta_i))^2] = MSE \tag{5}$$

2.5 Game & agent implementation

The snake game was implemented in Python using the `Pygame` package, where the game was played on a 640x480 window. The snake and bits of food were presented as rectangles of 20 pixels. At the start of each game, the snake was initialised in the middle of the window, whereas the bits of food were placed randomly (Appendix A). Each time the head of the snake collided with the food bit, a rectangle was added to the back of the snake and a new bit of food was placed randomly. The game was over once the snake's head collided with either a wall or its own body.

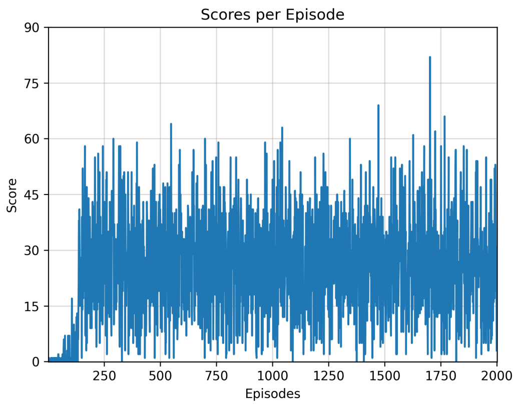
For the setup of the DQN, a feed-forward network consisting of an input layer, one hidden layer, and an output layer was constructed. Since the snake was able to move in three directions only (being left, straight ahead, or right, respectively) at each game iteration, the number of actions and thus the size of the output layer was set to three. Now, at the start of each iteration in the game, the snake could encounter a total of eleven different game states. These indicated a hazard ahead, left or right, its current direction either up, down, left or right, and if there was food in either of these four directions. Hence, the input layer size of the network was set to eleven. Finally, the dataset was expected to be quite large, so a conservative approach was used where it was opted for a number of 256 neurons for the hidden layer, a batch size of 1000 items and memory size of 100.000 items. [11]

2.6 Agent performance evaluation

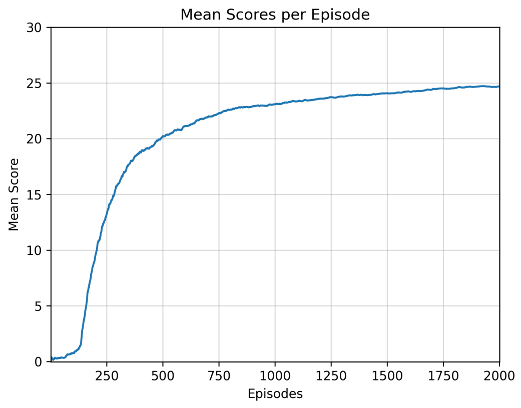
Now, for evaluating the agent's performance, the agent was tasked to play a number of 2000 episodes. This number was chosen to limit the agent's computational time consumption to match time constraints imposed on this project. Furthermore, ϵ was set to start from a value of 1 and was decayed to a value of 0.01 with a decay rate of 0.95 each episode. The learning rate α and discount factor γ were defaulted to values of 0.001 and 0.90, respectively, such that the relatively conservative approach in the setup of the neural network was matched. After completing the initial session, the resulting model with learned policies was saved. The performance of this model was then further evaluated by investigating the effects of varying parameters such as the probability decay rate, discount factor, learning rate, number of neurons in hidden layer, and replay memory batch size. Note that each parameter was adjusted individually, that is, whilst keeping the others constant, to ensure proper analysis. The performance evaluations themselves were conducted based on mean scores only as these indicated trends and overall agent behaviour in a way most suitable way for analysis. [4]

3 Results

Using the initial parameter settings and neural network architecture, the agent conducted its initial session consisting of 2000 episodes. The relevant agent's scoring pattern, as well as the relating mean score it achieved during this initial session, are presented in Figure 5a and Figure 5b, respectively.



(a) Scores achieved



(b) Mean scores achieved

Figure 5: Initial agent performance

As can be seen in Figure 5, in the first 150 episodes when the agent was still exploring, only low scores were achieved. Here, the agent would often bump into either walls or itself, while occasionally finding food. Past this point, the agent rapidly learned to avoid negative rewards and to focus on searching for food, as is indicated by the rapid increase in mean scores (Figure 5b). Past episode 750, the agent (which now had transferred to its exploitation phase) did increase its scoring performance, albeit at reduced rate, which indicated signs of stabilisation. Nearing the end of the session, the agent's performance showed more signs of consistency with the increase in mean scores achieved further flattening out. Therefore, the agent demonstrated its capability of handling the snake game well. It achieved scores over 50 frequently and even a peak score of 84. Interestingly, the phenomenon of scoring consistency was not yet fully visible in Figure 5a. Here, the fluctuations in scores did reduce only to minor extent, suggesting that full convergence of the agent model had not yet been achieved. The main limiting factor was found to be the number of episodes used, a result of time constraints imposed on the project. For more converged results, it is therefore recommended to further extend the number of episodes.

3.1 Effects of epsilon decay rate

To study implications epsilon decay rates have on the agent's performance, the agent conducted sessions with decay rates ranging from 0.94 up to 0.99. The resultant agent performances are presented in Figure 6.

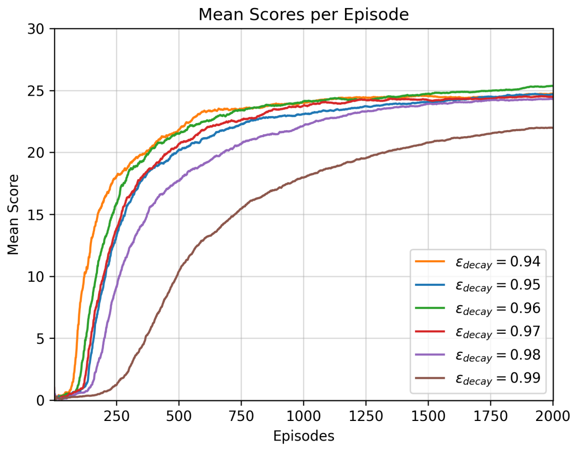


Figure 6: Mean scores achieved with varying epsilon decay rates

From the agent performances it was deduced that a slower epsilon decay rate imposed slower learning. This was an expected result, as the ϵ parameter dictated the agent's balance between exploration and exploitation. Reducing decay speed through larger values for decay rates therefore led to a prolonged exploration phase. This observation is substantiated by the agent's performance with a decay rate of 0.99, where slower learning was showcased by the reduced increase in mean scores. At the same time, the slower decay rate also implied a delay of convergence to a stable policy, as indicated by mean scores not yet fully stabilised near the session's end.

Conversely, imposing a faster decay rate resulted in more rapid learning due to reduced exploration time, as especially demonstrated by the decay rate of 0.94. However, it also demonstrated that it had actually attained a sub-optimal strategy. Consequently, the agent had converged prematurely, and so the scores it scored left room for improvement. Ultimately, the decay rate that was found to work best for the agent was 0.96. With this rate, the agent was able to strike its best balance between exploration and exploitation, resulting in the overall best performance. Note again that the agent with this setting was also not yet fully converged near the session's end, similar to other performances. Therefore, should sessions be extended, the relevant optimal decay rate may actually be slower.

3.2 Effects of discount factor

After resetting the parameters back to their original values, the agent was subjected to varying discount factors γ. Since this influenced to what extent the agent prioritised future rewards, it was expected to see different performances over 2000 episodes. When considering Figure 8, it is observed that this was indeed the case.

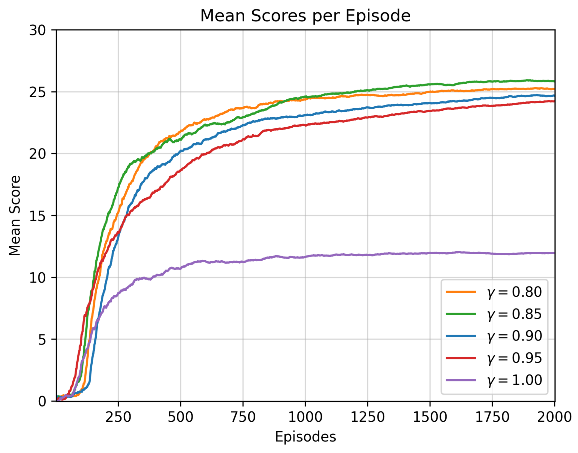


Figure 7: Mean scores achieved with varying discount factors

Figure 7 showed that decreasing the discount factor caused the agent to learn more rapidly and to achieve higher scores. Furthermore, it was found that the agent was controlling the snake more aggressively. Consequently, the agent was quicker at finding bits of food and thereby increasing its score. This can also be observed in Figure 8, where scoring performance was enhanced earlier and more rapidly relative to other sessions. It should be noted, however, that with this setting the agent struggled more with collecting bits of food located near the edges of the window. Nevertheless, since the majority of the food bits were not placed near edges, the agent's performance increase was only limited slightly and so setting the factor to 0.85 led to better performance.

The same could not be said about the agent that was playing with larger discount factors. When the agent was playing with such discount factors, it was observed that the learning progress was much slower. Additionally, the agent was found to exhibit much more risk-avoiding behaviour, which resulted in the agent tending to stay away from the edges of the playing field. Since a larger discount factor implies emphasis of future rewards, it was concluded that the agent's playing behaviour was overly cautious. Consequently, the agent did not seem able to adapt to short-term challenges imposed by placing bits near the edges of the window, leading to a relatively worsened and unfavourable performance.

3.3 Effects of learning rate

Subsequent to restoring the agent back to its initial form, the agent was subjected to varying learning rates α, which had been increased in steps of 0.0025. The corresponding agent performances are presented in Figure 8.

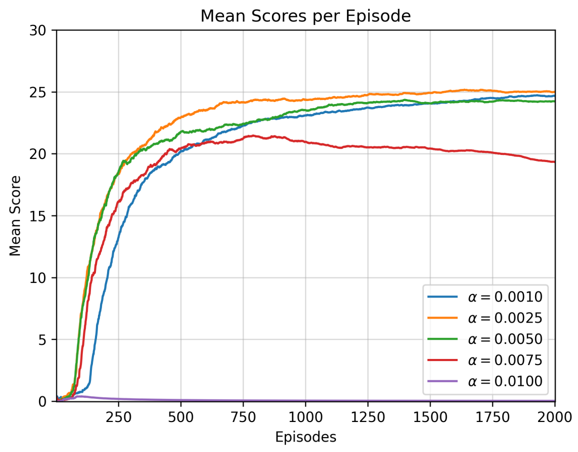


Figure 8: Mean scores achieved with varying learning rates

Similar to the epsilon decay rate results, an overall trend with increasing learning rates was immediately deduced. It was found that, when increasing the learning rates slightly, say to 0.0025, the agent was capable to learn faster and even attain a higher peak performance earlier than the agent with the initial settings did. The performance results for a learning rate of 0.0025 do show minor fluctuations, which indicated minor instabilities. These instabilities emerged more in the performance of the agent with a learning rate of 0.0050, as can be seen by the increase in fluctuations. Interestingly, the agent's learning rate was still increasing and the scoring performance was still decent despite the increase in instabilities.

When increasing the learning rate even further, it was found that the learning rate was simply too rapid for the agent. In the case of a learning rate of 0.0075, this resulted in a notable increase in instability which led to the agent overshooting what were the optimal Q-values. Consequently, a relative degradation of the agent's performance as the session progressed was observed. When then subjecting the agent to a learning rate of 0.0100, it was found that the learning rate was simply too high, as the agent failed to learn anything meaningful from its environment. Thus in this case, the model was found to diverge from what were good strategies and so was considered extremely unstable. Ultimately, the learning rates of 0.0010 and 0.0025 were both considered the most favourable for the task at hand, where the former could be chosen for smooth convergence and the latter for faster learning.

3.4 Effects of hidden neurons

After varying the agent's parameters, the model's neural network architecture was adjusted to evaluate the model's influence on agent performance. For this, the size of hidden layer was adjusted by varying the number of hidden neurons. Figure 9 presents the performances achieved with different hidden layer sizes.

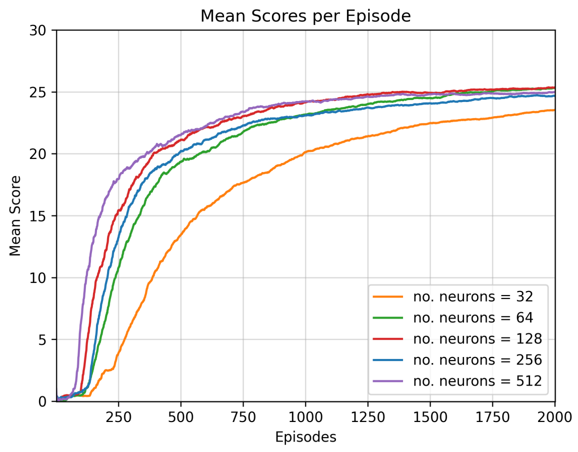


Figure 9: Mean scores achieved with different hidden layer sizes

Since the size of a hidden layer determined to what extent the model was able to identify complex patterns, a relationship between number of neurons and model capacity was anticipated. When observing Figure 9, such relationship was indeed found present. For the lower number of neurons, 32 and 64, respectively, it was observed that computational times were relatively lower. However, the gains in computational time came at a cost of learning speed and overall performance, especially for 32 hidden neurons. Here, the model capacity was deemed too limited, resulting in very slow learning and overall under-fitting of the data. Increasing the number of neurons to 64 did increase the agent's learning rate, though its moderate capacity still resulted in lower performance compared to the larger layer sizes and so, potentially, was still under-fitting data.

Further increasing the number of neurons to 128 produced a notable improvement in both the agent's learning rate and overall performance. In fact, the performance was even found to be quite similar to that of the initial model consisting of 256 neurons. The initial model did seem to generate a bit more generalised overall performance, though despite this the differences between the 128 neuron model and the 256 neuron model were still considered only marginal. Lastly, expanding the hidden layer to 512 neurons did allow the agent to learn relatively more rapidly initially, whilst achieving good scores overall. However, the performance improvements were not considered noteworthy, and therefore it was concluded that further addition of neurons was not relevant.

3.5 Effects of batch size

Lastly, the batch sizes of the replay memory batches were varied to study the influences the batch size has on updating the Q-values and thereby the agent learning dynamics. For this, the initial batch size was both reduced and increased twice, resulting again in the performances shown in Figure 10.

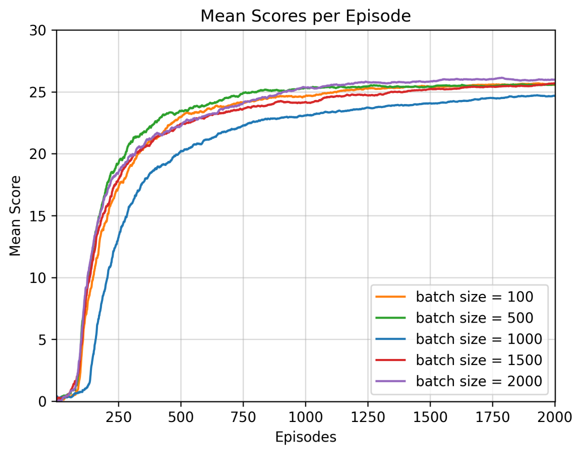


Figure 10: Mean scores achieved with different memory batch sizes

When initially reducing the batch size, it was found that the agent was able to learn the game more quickly than it originally did. However, similar to the observations with varying the learning rates, this did come at the cost of increased variances and thus increased instability. Furthermore, the agents in these cases also converged sooner than the agents with larger batch sizes did. This contributed to them converging to marginally lower peak performances as can be observed in Figure 10.

Surprisingly, increasing the batch sizes beyond the initial batch size also allowed for the agent to learn relatively quicker. In these cases, a better peak performance was obtained, albeit not fully converged. This may be explained by the fact that larger batch sizes allow for more reliable updates, which led to better performance improvements. As such, an interesting observation regarding the initial batch size of a 1000 items was made. Clearly, this batch size was neither large enough for attaining relatively better peak performance nor small enough for capitalisation on quick learning of smaller batches. Since this batch size did not strike a balance between either, it was concluded that for better performances larger batch sizes are favourable.

4 Discussion

The results of this study provide insights into the application of Deep Q-Learning to the classic Snake game. The performance of the AI agent, evaluated through various parameters, demonstrates both potential and limitations of reinforcement learning in dynamic environments which involves complex decision-making. One main finding is the impact of the epsilon decay rate on the agent's learning process. A slower decay rate prolonged the exploration phase, leading to slower learning with more exploration of the environment. Conversely, a faster decay rate increased the learning speed but at the risk of premature convergence to sub-optimal strategies. The optimal balance is found at an epsilon decay rate of 0.96, though even with this setting the agent had not fully converged, suggesting that further exploration could yield even better results.

The discount factor also played a crucial role in shaping the agent's behaviour. Lower discount factors caused more aggressive gameplay, but at the cost of the agent struggling near the edges of the game window. Higher discount factors emphasising long-term rewards led to more cautious behavior and came at the cost of slower learning and less optimal performances. Similarly, the learning rate also had an evident influence on the agent's performance. A moderate learning rate allowed the agent to learn efficiently and achieve higher scores earlier, whereas increasing the learning rate too much resulted in instability and, eventually, divergence. These findings underline the importance of tuning parameters carefully to strike a balance between learning speed and stability. Furthermore, adjusting the hidden layer size in the neural network revealed that, while increasing the number of neurons did improve the agent's learning capacity, the performance gains were marginal. Specifically, the difference between models with 128 and 256 neurons was minimal, suggesting that for this particular task, a smaller model could suffice. Lastly, varying batch sizes revealed that smaller batches allowed for quicker but more unstable learning, where on the other hand larger ones provided more reliable updates and better overall performance.

A notably significant constraint in this study was the limitation of the hardware, which restricted the number of episodes to 2000 due to the project's time constraints. This limitation had several implications for the results and their interpretation. Firstly, the restriction to 2000 episodes prevented the agent from completely converging to optimal strategies. Since in reinforcement learning convergence requires a large number of episodes, especially in complex environments like the Snake game, the limited number of episodes likely resulted in the agent only partially learning optimal strategies. This could have contributed to the observed fluctuations in performance and general lack of complete stabilisation. Additionally, the limited episode number constrained the ability to fully evaluate "long-term effects" of parameter changes. For instance, slower epsilon decay rates or higher discount factors might have shown greater benefits over a more extended training period. Similarly, larger batch sizes and networks with more hidden neurons could have yielded better performance if given more time to train. The time constraints imposed by hardware limitations thus imply that conclusions drawn from this study, while valuable, are not fully indicative of the agent's potential performance if trained for longer periods.

In summary, while the study demonstrates the application of Deep Q-Learning to the Snake game and provides insights into parameter tuning, the hardware limitations impacted the depth of analysis. Future studies could therefore benefit from access to more powerful hardware, potentially leading to enhanced robustness of conclusions and, potentially, even better-performing agents.

5 Conclusion

This study explored application of Deep Q-Learning to the Snake game and analysed how tuning parameters affect the agent's performance. The study focused on evaluating the epsilon decay rates, discount factors, learning rates, hidden neurons, and batch sizes. The results showed that the epsilon decay rate plays a crucial role in balancing exploration and exploitation, with a decay rate of 0.96 yielding the best results. The discount factor influenced the agent's behavior as well, where lower values encouraged aggressive gameplay but also led to sub-optimal strategies. The learning rate was also found to contribute significantly, where smaller values provided a good balance between learning speed and stability and larger ones exposed the model's instability.

Furthermore, when adjusting the neural network's architecture, it was found that, while increasing hidden neurons improved performance, the benefits diminished beyond a certain point. Lastly, smaller batch sizes were found to generate faster but more unstable learning, whereas larger ones generated a better balance between speed and stability. The initial batch size setting was interestingly found to not benefit from either of these factors. Important is, however, that the study faced limitations imposed by hardware constraints, restricting the sessions to 2000 episodes. This limitation likely prevented the agent from completely converging to optimal strategies. The limited number of episodes also restricted the exploration of long-term effects of parameter adjustments, suggesting that the agent's full potential may not have been reached within this study and that further studies are necessary.

In conclusion, while the study provides useful insights into applying Deep Q-Learning to the Snake game and its parameter tuning, the hardware constraints likely limited the agent's performance. Future research should therefore focus on overcoming these hardware limitations, such that the agent may be exposed to extended sessions and such that more robust results can be generated, potentially leading to even more effective agents.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[3] J. Norris, *Markov Chains.* Cambridge: Cambridge University Press, 1998.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press, 2nd ed., 2018.

[5] Y. Li, "Deep reinforcement learning," *Cornell University ArXiv*, 2018.

[6] G. Tesauro, "Temporal difference learning and td-gammon," *Commun. ACM*, vol. 38, p. 58–68, mar 1995.

[7] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[8] Z. Yang, Y. Xie, and Z. Wang, "A theoretical analysis of deep q-learning," *Cornell University arXiv*, 2019.

[9] G. Cybenko, "Approximation by superpositions of a sigmoidal function. math cont sig syst (mcss) 2:303-314," *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303–314, 12 1989.

[10] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* Cambridge, MA, USA: MIT Press, 2016.

[11] P. Loeber, "Train an ai to play snake using python - reinforcement learning with pytorch." Published by FreeCodeCamp, 2020.

A Snake Game Window

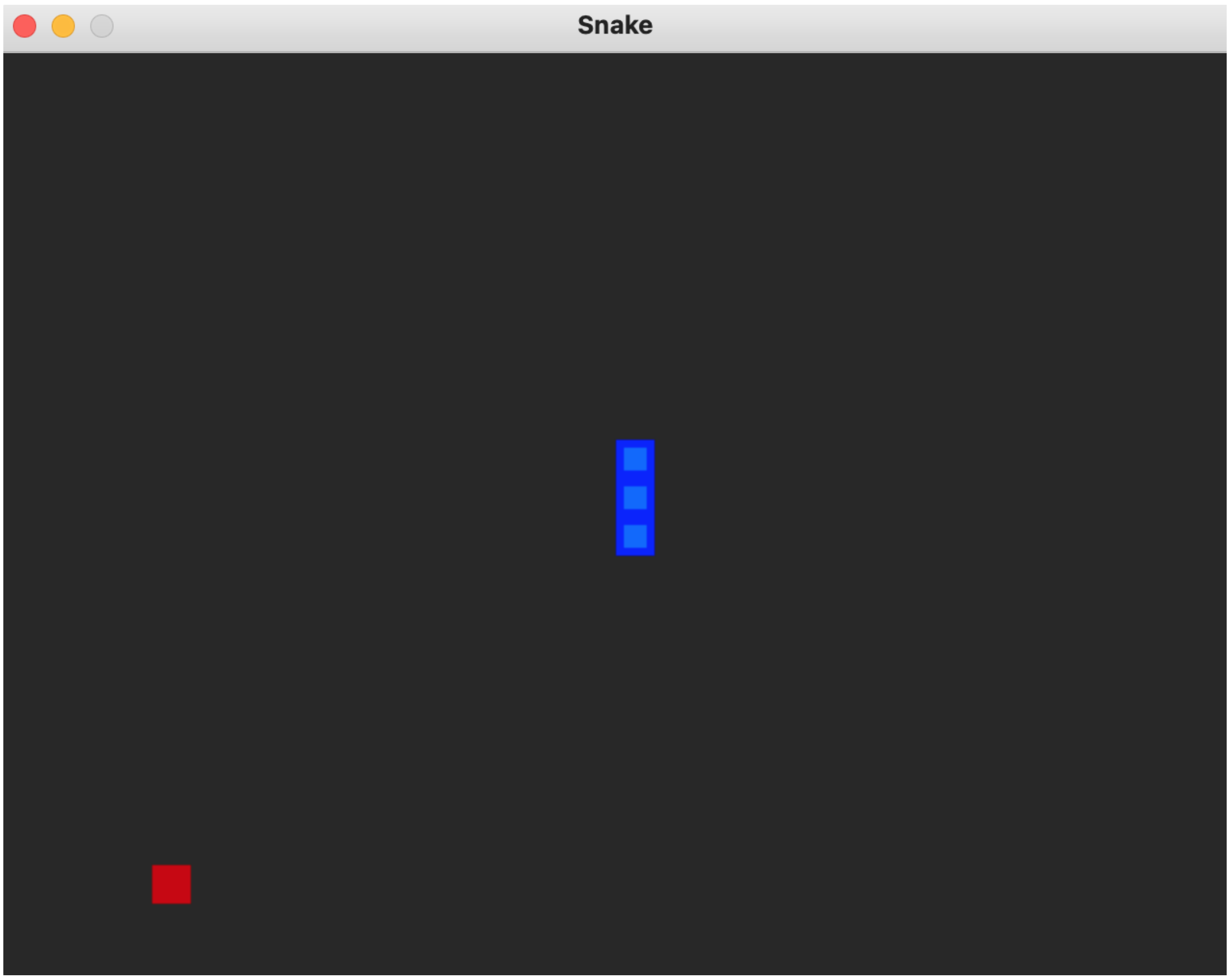


Figure 11: Snake Game Window Setup